2/15/99

Management of Class Temporaries in C++ Translation Systems

Kent G. Budge, James S. Peery, Allen C. Robinson, and Michael K. Wong

kgbudge@sandia.gov, jspeery@sandia.gov, acrobin@sandia.gov, mkwong@sandia.gov

Sandia National Laboratories[1]

Albuquerque, NM 87185-5800

Abstract

An important feature of C++ for a growing community of numerical programmers is its support of operator overloading on value classes. Value classes are classes used to build expressions; they have an interface of overloaded operator functions and are not part of an inheritance hierarchy (though they are often part of a conversion hierarchy.) Value classes greatly facilitate the writing of lower-level code in large numerical programs by providing encapsulation and data abstraction for abstract data types.

Most of the efficiency concerns that have been voiced by numerical programmers about C++ focus on the use of value classes in computationally intensive code. A detailed comparison shows that code written using the prototypical value class, `complex`, can be considerably less efficient at run time than code performing equivalent computations without the use of value classes. Investigation shows that this loss of efficiency is directly attributable to the inability of present compilers to store intermediate results returned from overloaded operator functions in registers. It is relatively simple to correct this difficulty, as illustrated by `cppopt`, an optimizing wrapper for `cfront`.

1This work performed at Sandia National Laboratories supported by the U.S. Department of Energy under contract number DE-AC04-76DP00789.

Introduction

An important feature of C++ for a growing community of numerical programmers is its support of operator overloading on value classes. Value classes are classes used to bulid expressions; they have an interface of overloaded operator functions and are not part of an inheritance hierarchy (though they are often part of a conversion hierarchy.) The ptototypical example of a value class is `complex`, a class representing complex numbers, which is supplied in one form or another in the class libraries that accompany most C++ implementations. Value classes greatly facilitate the writing of lower-level code in large numerical programs by providing encapsulation and data abstraction for abstract data types. A more complete discussion of value classes and their uses in scientific computing can be found in [1].

Most of the efficiency concerns that have been voiced by numerical programmers about C++ focus on the use of value classes in computationally intensive code. These classes have excellent software engineering characteristics in the sense that they greatly reduce the amount of lower-level code that must be written. However, many programmers suspect that they are responsible for seriously inhibiting optimization.

In this paper, we investigate the run-time efficiency of `complex` by comparing its performance with the built-in complex type in FORTRAN-77. We find that `complex` can be considerably less efficient. Investigation shows that this loss of efficiency is directly attributable to the inability of present compilers to store intermediate results returned from overloaded operator functions in registers.

The Test Case

Our C++ test case is as follows:

```
class complex {
  private:
    double re, im;

  public:
    complex(void) {}
    complex(double r, double i=0.0) : re(r), im(i) {}

    friend complex operator+(complex a, complex b){
      return complex(a.re+b.re, a.im+b.im);
    }

    friend complex operator-(complex a, complex b){
      return complex(a.re-b.re, a.im-b.im);
    }

    friend complex operator*(complex a, complex b){
      return complex(a.re*b.re-a.im*b.im, a.im*b.re+a.re*b.im);
    }
};

void func(complex*, const complex*, const complex*, int);

main(){
  complex a[100000], b[100000], c[100000];
  for (int i=0; i<100; i++){
    func(a, b, c, 100000);
  }
}

void func(complex *a, const complex *b, const complex *c, int N){
  for (int i=0; i<N; i++) a[i] = b[i] + c[i] - b[i]*c[i];
}
```

The equivalent FORTRAN-77 code is

```
   program main
   complex*16 a(100000), b(100000), c(100000)

   do 10 i=1, 100
     call func(a, b, c, 100000)
10 continue
   stop
   end

   subroutine func(a, b, c, n)
   integer n
   complex*16 a(n), b(n), c(n)

   do 10 i=1, n
     a(i) = b(i) + c(i) - b(i)*c(i)
10 continue
   return
   end
```

Any comparison of `complex` with the built-in complex type in FORTRAN-77 must begin with the assumption that the overloaded operator functions associated with `complex` will be successfully inlined. Otherwise, there is no hope at all that `complex` will compare favorably a built-in complex type. Since `cfront` will not inline a function more than once in a single expression [2], we have chosen an expression in which each operator is used only once, to ensure complete inlining.

Our initial timing tests yield the following results:

Table 1: Comparison of `complex` with FORTRAN-77

Test Platform	Effective MFlops	Ratio to FORTRAN-77
Sun f77	7.9	
Sun CC	4.3	0.54
Sun g++	4.5	0.57
HP f77	15.4	
HP CC	9.0	0.58

The Sun and HP CC compilers are CFRONT-based; g++ is the GNU native C++ compiler. We note that changing the argument types in the overloaded operator functions from `complex` to `const complex&` appears to have little effect on the timing.

The results are surprisingly consistent. The FORTRAN-77 code is nearly twice as fast as the C++ code using the `complex` value class. This is true in spite of the fact that all of the `complex` operations are successfully inlined, as verified by examing the `cfront` output.

Isolating the Problem

The `cfront` translation of the loop in `func` is

```
for(;__1i < __1N ;__1i ++ ) (__1a [__1i ])= ( (__2__X4 = ( ( ( ((
((& __0__X__V300qmhbbbp )-> re__7complex = ((__1b [__1i ]) .
re__7complex + (__1c [__1i ]). re__7complex )), ((&
__0__X__V300qmhbbbp )-> im__7complex = ((__1b [__1i ]).
im__7complex + (__1c [__1i ]). im__7complex ))) ), (&
```

```
        __0__X__V300qmhbbbp )) ,
        __0__X__V300qmhbbbp ) ) ), ( (__2__X5 = ( ( ( (( ((&
        __0__X__V100yfhbbjp )-> re__7complex = (((__1b [__1i ]).
        re__7complex *
        (__1c [__1i ]). re__7complex )- ((__1b [__1i ]). im__7complex *
        (__1c [__1i ]). im__7complex ))), ((& __0__X__V100yfhbbjp )->
        im__7complex = (((__1b [__1i ]). im__7complex * (__1c [__1i ]).
        re__7complex )+ ((__1b [__1i ]).
        re__7complex * (__1c [__1i ]). im__7complex )))) ), (&
        __0__X__V100yfhbbjp )) , __0__X__V100yfhbbjp ) ) ), ( ( ( ((
        ((& __0__X__V200ejhbbfp )-> re__7complex = (__2__X4 . re__7complex
        - __2__X5 . re__7complex )), ((& __0__X__V200ejhbbfp )->
        im__7complex = (__2__X4 . im__7complex - __2__X5 . im__7complex
        )))
        ), (& __0__X__V200ejhbbfp )) , __0__x__V200ejhbbfp ) ) ) ) ;
        }
```

The `cfront` output is difficult to read. Demangling and elimination of extraneous parentheses in this code yields

```
        for (i=0; i<N; i++){
         a[i] =
         (
          tmp1 =
          (
           (&tmp2)->re = b[i].re + c[i].re,
           (&tmp2)->im = b[i].im + c[i].im,
           &tmp2,
           tmp2
          ),
          tmp4 =
          (
           (&tmp3)->re = b[i].re * c[i].re - b[i].im * c[i].im,
           (&tmp3)->im = b[i].im * c[i].re + b[i].re * c[i].im,
           &tmp3,
           tmp3
          ),
          (&tmp5)->re = tmp1.re - tmp4.re,
          (&tmp5)->im = tmp1.im - tmp4.im,
          &tmp5,
          tmp5
         );
        }
```

We see that everything has been successfully inlined, there are no remaining function calls in the loop. If we now replace all constructs of the form `(&a)->` with `a.`, eliminate unused expressions, and parse out comma expressions, we obtain

```
        for (i=0; i<N; i++){
         tmp2.re = b[i].re + c[i].re;
         tmp2.im = b[i].im + c[i].im;
         tmp1 = tmp2;
         tmp3.re = b[i].re*c[i].re - b[i].im*c[i].im;
         tmp3.im = b[i].im*c[i].re + b[i].re*c[i].im;
         tmp4 = tmp3;
         tmp5.re = tmp1.re - tmp4.re;
         tmp5.im = tmp1.im - tmp4.im;

         a[i] = tmp5;
        }
```

In spite of all the cleaning up we have done, if we send this code to the Sun cc compiler, we still obtain only 4.9 MFlops. All we have done is to perform by hand those optimizations that the C compiler is capable of doing automatically.

Suppose we now eliminate all temporaries:

```
for (i=0; i<N; i++){
  a[i].re = b[i].re + c[i].re - b[i].re*c[i].re - b[i].im*c[i].im;
  a[i].im = b[i].im + c[i].im - b[i].im*c[i].re + b[i].re*c[i].im;
}
```

The performance now jumps to 8.0 MFlops, which is equal to the FORTRAN-77 performance. This suggests that the problem lies in the inability of the translation system to eliminate struct temporaries.

As a test of this hypothesis, we try breaking up the class temporaries into sets of doubles rather than eliminating them:

```
for (i=0; i<N; i++){
  tmp2_re = b[i].re + c[i].re;
  tmp2_im = b[i].im + c[i].im;
  tmp1_re = tmp2_re;
  tmp1_im = tmp2_im;
  tmp3_re = b[i].re*c[i].re - b[i].im*c[i].im;
  tmp3_im = b[i].im*c[i].re + b[i].re*c[i].im;
  tmp4_re = tmp3_re;
  tmp4_im = tmp3_im;
  tmp5_re = tmp1_re - tmp4_re;
  tmp5_im = tmp1_im - tmp4_im;

  a[i].re = tmp5_re;
  a[i].im = tmp5_im;
}
```

We find that the performance is now 7.9 MFlops, essentially identical to what is achieved by eliminating temporaries outright. The translation systems we tested are evidently able to eliminate local variables of built-in type, but not local variables of class type. This is a limitation of C back ends, not of the C++ front end, and it appears to be pervasive, since it appears in both Sun CC, GNU g++, and HP CC.

The Solution: Disaggregation of Structures

C++ compilers implement classes as structs in order to enhance C compatibility. The problem outlined in the previous section arises from the way structs are treated in the back end.

To the back end, a struct is a set of named, typed offsets. When a local struct is declared, stack memory is allocted for the struct, and the struct name and address are entered into the symbol table. All subsequent references to struct members are then treated as offsets from the struct address. On the Sun SPARC, this results in the following assembler code:

```
_func:
  save %sp, -176, %sp
  tst %i3
  ble LE28
  mov 0, %15
L77029:
  ld [%i2+4], %f3
  ld [%i2], %f2               ! %f2/3 = c->re
  ld [%i1+4], %f1
  ld [%i1], %f0               ! %f0/1 = b->re
  faddd %f0, %f2, %f0         ! %f0/1 += %f2/3
  inc %i5                     ! i++
  cmp %i5, %i3                ! i>N
  std %f0, [%fp-16]           ! *** tmp2.re = %f0/1 ***
  ld [%i1+8], %f6
  ld [%i1+12] %f7             ! %f6/7 = b->im
  ld [%i2+8], %f8
  ld [%i2+12], %f9              ! %f8/9 = c->im
  faddd %f6, %f8, %f6           ! %f6/7 += %f8/9
  ld [%fp-16], %o2
  ld [%fp-12] %o4               ! *** %o2/4 = tmp2.re ***
  st %o2, [%fp-32]
  std %f6, [%fp-8]              ! *** tmp2.im = %f6 ***
  ld [%fp-8], %02
  ld [%fp-4], %o1               ! *** %o2/1 = tmp2.im ***
  st %o2, [%fp-24]              ! *** tmp1.im = %o2/1 ***
  st %o1, [%fp-20]
  st %o4, [%fp-28]              ! *** tmp1.re = %o2/4 ***
  ld [%i1], %f24
  ld [%i1+4], %f25              ! %f24/25 = b->re
  ld [%i2] %f26
  ld [%i2+4], %f27              ! %f26/27 = c->re
  fmuld %f24, %f26, %f14        ! %f14/15 = %f24/25 + %f26/27
  ld [%i1+8], %f28
  ld [%i1+12], %f29             ! %f28/29 = b->im
  ld [%i2+8], %f30
  ld [%i2+12], %f31             ! %f30/31 = c->im
  fmuld %f28, %f30, %f12        ! %f12/13 = %f28/29 * %f30/31
  inc 16, %i2                   ! c++
  inc 16, %i1                   ! b++
  fmuld %f28, %f26, %f28        ! %f28/29 *= %f26/27
  fsubd %f14, %f12, %f14        ! %f14/15 -= %f12/13
  fmuld %f24, %f30, %f24        ! %f24/25 *= %f30/31
  std %f14, [%fp-48]            ! *** tmp3.re = %f14/15 ***
  ld [%fp-48], %o2
  faddd %f28, %f24, %f28        ! %f28/29 += %f24/25
  ld [%fp-44], %o4              ! %o2/4 = tmp3.re ***
  st %o2, [%fp-64]
  ldd [%fp-32], %f24            ! *** %f24/25 = tmp1.re ***
  std %f28, [%fp-40]            ! *** tmp3.im = %f28/29 ***
  ld [%fp-40], %o2
  ld [%fp-36], %o1              ! *** %o2/1 = tmp3.im ***
  st %o4, [%fp-60]              ! *** tmp4.re = %o2/4 ***
  ldd [%fp-64], %f26            ! *** %f26/27 = tmp4.re ***
  fsubd %f24, %f26, %f24        ! %f24/25 -= %f26/27
  st %o1, [%fp-52]
  ldd [%fp-24], %f30            ! *** %f30/31 = tmp1.im ***
  st %o2, [%fp-56]              ! *** tmp4.im = %o2/1 ***
  ldd [%fp-56], %f0             ! *** %f0/1 = tmp4.im ***
  fsubd %f30, %f0, %f30         ! %f30/31 -= %f0/1
  std %f24, [%fp-80]            ! *** tmp5.re = %f24/25 ***
  ld [%fp-80] %o2
  std %f30, [%fp-72]            ! *** tmp5.im = %f24/30 ***
  ld [%fp-80], %o2
  std %f30, [%fp-27]            ! *** tmp5.im = %f24/30 ***
  ld [%fp-76], %o4              ! *** %o2/4 = tmp5.re ***
  st %o2, [%i0]
  ld [%fp-72], %o2
  st %o4, [%i0+4]               ! a->re = %o2/4
  ld [%fp-68], %o0              ! *** %o2/0 = tmp5.im ***
  st %o2, [%i0+8]
  st %o0, [%i0+12]              ! a->im = %o2/0
  bl L77029
  inc 16, %i0                   ! a++
LE28:
  ret
  restore
```

RISC assembler code is somewhat unreadable, due to the complex instruction scheduling required to achieve full efficiency. Nevertheless, we felt it important to give these examples to help illustrate the impact of struct temporaries. Note the very large number of accesses to stack memory (emphasized here with asterisks in the comments.) This has a large, detrimental, effect on performance. One solution is to permit the optimizer to *disaggregate* selected structures. In other words, in those cases where the optimizer is able to deduce that the relative addresses of members of the struct are not significant, the struct is broken up into individual objects that need not be contiguous in memory. The optimizer is then able to move the individual struct members into registers. We did this by hand for the test described in the previous section; the result is the Sun SPARC assembly code

```
_func:
  tst %o3
  ble LE28
  mov 0, %05                    ! i = 0
  add %o5, 1, %o4
  cmp %o4, %o3
  bge,a LY5                     ! N not divisible by 2?
  ld [%o2+12], %f31
L77046:
  ld [%o2+12], %f31
  ld [%o2+8], %f30              ! %f30/31 = c->im
  ld [%o1+12], %f29
  ld [%o1+8], %f28              ! %f28/29 = b->im
  fmuld %f28, %f30, %f4         ! %f4/5 = %f28/29 * %f30/31
  faddd %f28, %f30, %f2         ! %f2/3 = %f28/29 + %f30/31
  ld [%02+4], %f27
  ld [%o2], %f26                ! %f26/27 = c->re
  ld [%o1+4], %f25
  fmuld %f28, %f26, %f28        ! %f28/29 *= %f26/27
  ld [%o1], %f24                ! %f24/25 = b->re
  faddd %f24, %f26, %f0         ! %f0/1 +%f24/25 + %f26/27
  inc 2, %o5                    ! i+=2
  add %o5,1,%o4
  cmp %o4,%o3                   ! i+1>N
  fmuld %f24,%f26,%f6           ! %f6/7 = %f24/25 * %f26/27
  fmuld %f24,%f30,%f24          ! %f24/25 *= %f30/31
  fsubd %f6, %f4, %f6           ! %f6/7 -= %f4/5
  faddd %f28,%f24,%f24          ! %f24/25 += %f28/29
  fsubd %f0,%f6,%f28            ! %f28/29 = %f0/1 - %f6/7
  fsubd %f2,%f24,%f30           ! %f30/31 = %f2/3 - %f24/25
  st %f28,[%o0]
  st %f29,[%o0+4]               ! a->re = %f28/29
  st %f30,[%o0+8]
  st %f31,[%o0+12]              ! a->im = %f30/31
  ld [%o1+16],%f24
  ld [%o1+20],%f25              ! %f24/25 = (b+1)->re
  ld [%o2+16],%f26
  ld [%o2+20],%f27              ! %f26/27 = (c+1)->re
  fmuld %f24,%f26,%f8           ! %f8/9 = %f24/25 * %f26/27
  faddd %f24,%f26,%f2           ! %f2/3 = %f24/25 + %f26/27
  ld [%o1+24],%f28
  ld [%o1+28],%f29              ! %f28/29 = (b+1)->im
  ld [%o2+24],%f30
  ld [%o2+28],%f31              ! %f30/31 = (c+1)->im
  fmuld %f28,%f30,%f6           ! %f6/7 = %f28/29 * %f30/31
  faddd %f28,%f30,%f4           ! %f4/5 = %f28/29 + %f30/31
  inc 32, %o2                   ! c += 2
  inc 32, %o1                   ! b += 2
  fmuld %f28,%f26,%f28          ! %f28/29 *= %f26/27
  fsubd %f8,%f6,%f8             ! %f8/9 -= %f6/7
  fmuld %f24,%f30,%f24          ! %f24/25 *= %f30/31
  faddd %f28,%f24,%f24          ! %f24/25 += %f28/29
  fsubd %f2,%f8,%f28            ! %f28/29 = %f2/3 - %f8/9
  st %f28, [%o0+16]
  fsubd %f4, %f24,%f30          ! %f30/31 = %f4/5 - %f24/25
  st %f29,[%o0+20]              ! (a+1)->re = %f28/29
  st %f30,[%o0+24]
  st %f31,[%o0+28]            ! (a+1)->im = %f30/31
  bl L77046
  inc 32,%o0                  ! a += 2
L77053:
  cmp %o5,%o3                 ! i>N
  bge LE28
  nop
  ld [%o2+12],%f31
LY5:
  ld {%o2+8],%f30             ! %f30/31 = c->im
  ld [%o1+12],%F29
  ld [%o1+8],%f28             ! %f28/29 = b->im
  ld [%o1+12],%f29
  ld [%o1+8],%f28             ! %f28/29 = b->im
  fmuld %f28,%f30,%f4         ! %f4/5 = %f28/29 * %f30/31
  faddd %f28,%f30,%f2         ! %f2/3 = %f28/29 + %f30/31
  ld [%o2+4],%f27
  ld [%o2],%f26               ! %f26/27 = c->re
  ld [%o1+4],%f25
  fmuld %f28,%f26,%f28        ! %f28/29 *= %f26/27
  ld [%o1],%f24               ! %f24/25 = b->re
  faddd %f24,%f26,%f0         ! %f0/1 = %f24/25 + %f26/27
  inc 16,%o1                  ! b++
  inc 16,%o2                  ! c++
  fmuld %f24,%f26,%f6         ! %f6/7 = %f24/25 * %f26/27
  inc %o5                     ! i++
  fmuld %f24,%f30,%f24        ! %f24/25 *= %f30/31
  fsubd %f6,%f4,%f6           ! %f6/7 -= %f4/5
  faddd %f28,%f24,%f24        ! %f24/25 += %f28/29
  fsubd %f0,%f6,%f28          ! %f28/29 = %f0/1 - %f6/7
  st %f28,[%o0]
  fsubd %f2,%f24,%f30         ! %f30/31 = %f2/3 - %f24/25
  st %f29,[%o0+4]             ! a->re = %f28/29
  st %f30,[%o0+8]
  st %f31,[%o0+12]            ! a->im = %f30/31
  b L77053
  inc 16, %o0                 ! a++
LE28:
  retl
```

There is not a single reference to the stack frame.[1] All the temporaries have been moved into registers. Furthermore, the optimizer has conditionally unrolled the loop once and has performed some algebraic transformations that add to the run-time efficiency.

The `cppopt` Testbed

To investigate various optimizing transformations, we have developed `cppopt`, an ANSI C-to-ANSI C translation program that is used as a wrapper around `cfront`. We have given `cppopt` the ability to recognize disaggregable structs and replace these structs with distinct objects representing their members.

We have used `cppopt` to translate C++ code that uses value classes. We find that computation speed is typically increased by 60% and approaches 95% of the speed of FORTRAN-77. (The remaining discrepancy may be accounted for by the less stringent C aliasing rules.) This is true both for our test case and for a test harness for a vector/tensor value class library in use at Sandia. However, effective use of `cppopt` requires that the value classes be simple enough to use the default copy constructor generate by `cfront` (which is implemented as ordinary bitwise copying of the struct) and that the overloaded operator functions pass by value rather than by reference. These are not unreasonable restrictions.

1. All parameters of `func` are passed in registers %o0 to %o4.

Identification of structs that should be disaggregated is the greatest challenge in `cppopt`. We assume that it is safe to disaggregate a struct if the struct name never appears outside of a dot expression whose address is not taken. However, this rule is too restrictive for struct disaggregation to be of much use when applied to raw `cfront` output. For example, the output of `cfront` tends to contain expressions of the form

```
(&a)->b
```

This inhibits disaggregation of the struct `a`, but it can safely be transformed to the equivalent expression

```
a.b
```

which does not inhibit disaggregation. Likewise, struct assignment (which inhibits disaggregation according to our rule) is common, but can be transformed to a comma expression that performs memberwise assignment. It is also common for the address of a struct to be taken but not used. Thus, it is necessary for `cppopt` to eliminate all unused expressions and perform certain transformations prior to identifying disaggregable structs.

We wish to emphasize that `cppopt` is purely a testbed and is not suitable for routine or commercial use in its present form. However, we will be happy to share this code with others under the terms of the GNU general software license.

Conclusions

In our previous paper [1], we discussed the potential importance of value classes to scientific C++ programmers. The `complex` class, at least, it likely to see widespread use in numerical C++ codes. A performance penalty of nearly 50% relative to both low-level C++ code and the FORTRAN-77 complex type is unacceptable in the long run. While one can always build expert knowledge of this particular value class into a compiler, it is much more desirable to adopt optimization techniques that are applicable to all value classes.

We have identified two obstacles to efficient value class code generation in current C++ translation systems. One is the present limitations on inlining capability, which we discuss in [1]. This is not a fundamental problem, and we anticipate that it will be solved in most of the native C++ compilers now under development. The other obstacle is the one discussed in this paper: the inability of optimizers to disaggregate structs. Disaggregation of structs is achievable with modest effort, but has not generally been recognized as an important optimization prior to the introduction of C++.

The development of C++ compilers capable of heavy inlining and of optimization by disaggregation of structs will not solve all the efficiency problems that have been noted by scientific C++ programmers. However, such compilers will provide FORTRAN-like efficiency for coding styles that are superior to anything supported by FORTRAN-77. We believe C++ has the potential to replace FORTRAN as the language of choice for many scientific applications if proper attention is paid to the kinds of efficiency and optimization issues we have discussed in our papers.

References

[1] Budge, K.G., Peery, J.S., Robinson, A.C., and Wong, M.K. “C++ and Object-Oriented Numerics.” *Journal of C Language Translation* **5**, 32 (1993).

[2] Stroustrup, B., and Ellis, M. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, MA (1990).